# Swagger API

## What is Swagger API?

Swagger is a set of open-source tools built around the **OpenAPI Specification (OAS)**. It provides a framework to design, build, document, and consume RESTful APIs efficiently. Swagger ensures that APIs are clearly defined, standardized, and easily consumable by developers, testers.

Automatically generates an interactive API documentation interface.

## Why Use Swagger API?

- **Interactive Documentation:** Swagger generates clear, interactive documentation, allowing users to test APIs directly within the documentation.
- **Ease of Collaboration:** Provides a common platform for backend developers, frontend developers, and QA teams.
- **Improved API Testing:** Enables real-time testing of endpoints directly from the Swagger UI.

### Postman vs. Swagger: A Comparison

| Aspect | Swagger | Postman |
|---|---|---|
| **Primary Purpose** | API design, documentation, and development. | API testing, debugging, and monitoring. |
| **Key Feature** | Generates interactive API documentation from code or spec. | Simplifies API testing and debugging workflows. |
| **Documentation** | Built-in with Swagger UI and OAS specification. | Documentation is manual; collections can be shared. |
| **Design vs Testing** | Focuses more on API **design and documentation**. | Focuses more on API **testing and automation**. |
| **Ease of Use** | Requires integration with code (e.g., annotations in Spring). | Easy to set up and use for manual or automated testing. |
| **Code Generation** | Generates server stubs and client SDKs from OpenAPI specs. | Does not generate server/client code. |

| | | |
|---|---|---|
| **Interactive UI** | Yes, interactive API testing via Swagger UI. | Yes, but more testing-focused with parameter handling. |
| **Team Collaboration** | Collaboration via shared API specs. | Collaboration via shared workspaces/collections. |
| **Monitoring and CI/CD** | Limited support for CI/CD workflows. | Excellent for API monitoring, CI/CD, and testing. |
| **OpenAPI Support** | Native support for OpenAPI Specification. | Can import/export OpenAPI specs but not native. |
| **Offline/Cloud** | Works locally or hosted via Swagger Hub. | Works offline with a desktop app and supports cloud. |

## Step 1: Add Dependency

Add the springdoc-openapi-starter-webmvc-ui dependency to your pom.xml:

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.6.0</version>
</dependency>
```

## Step 2: Create a Spring Boot Application

Generate a simple Spring Boot application with the following structure:

**1.Create a Spring Boot project with these dependencies:**

- Spring Web
- Spring Boot DevTools (optional, for live reload)
- springdoc-openapi-starter-webmvc-ui

2.Add the `@SpringBootApplication` class to run your application.

CountryLocationsApplication.java

```java
package com.locations;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class CountryLocationsApplication {

    public static void main(String[] args) {
        SpringApplication.run(CountryLocationsApplication.class, args);
    }

}
```

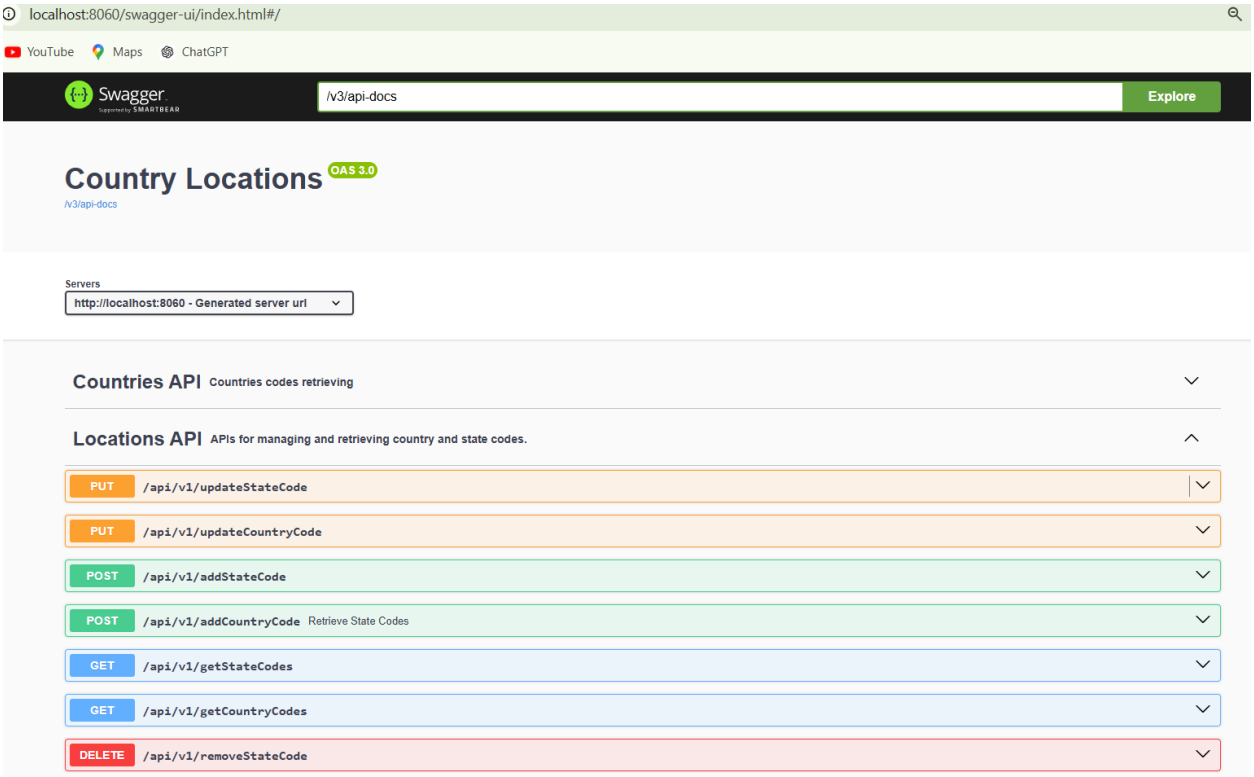
## Step 3: Create a REST Controller

```java
import java.util.HashMap;

@RestController
@RequestMapping("/api/v1")
@Tag(name = "Locations API", description = "**APIs for managing and retrieving country and state codes.**")
public class LocationsController {

    private final Map<String, String> countryCodes = new HashMap<>() {
        {
            put("US", "United States");
            put("IN", "India");
            put("CA", "Canada");
            put("UK", "United Kingdom");
        }
    };

    private final Map<String, String> stateCodes = new HashMap<>() {
        {
            put("CA", "California");
            put("NY", "New York");
            put("TX", "Texas");
            put("WA", "Washington");
        }
    };


    @GetMapping("/getCountryCodes")
    @Tag(name = "Countries API", description = "**Countries codes retrieving**")
    public ResponseEntity<Map<String, String>> getCountryCodes() {
        return ResponseEntity.ok(countryCodes);
    }
```

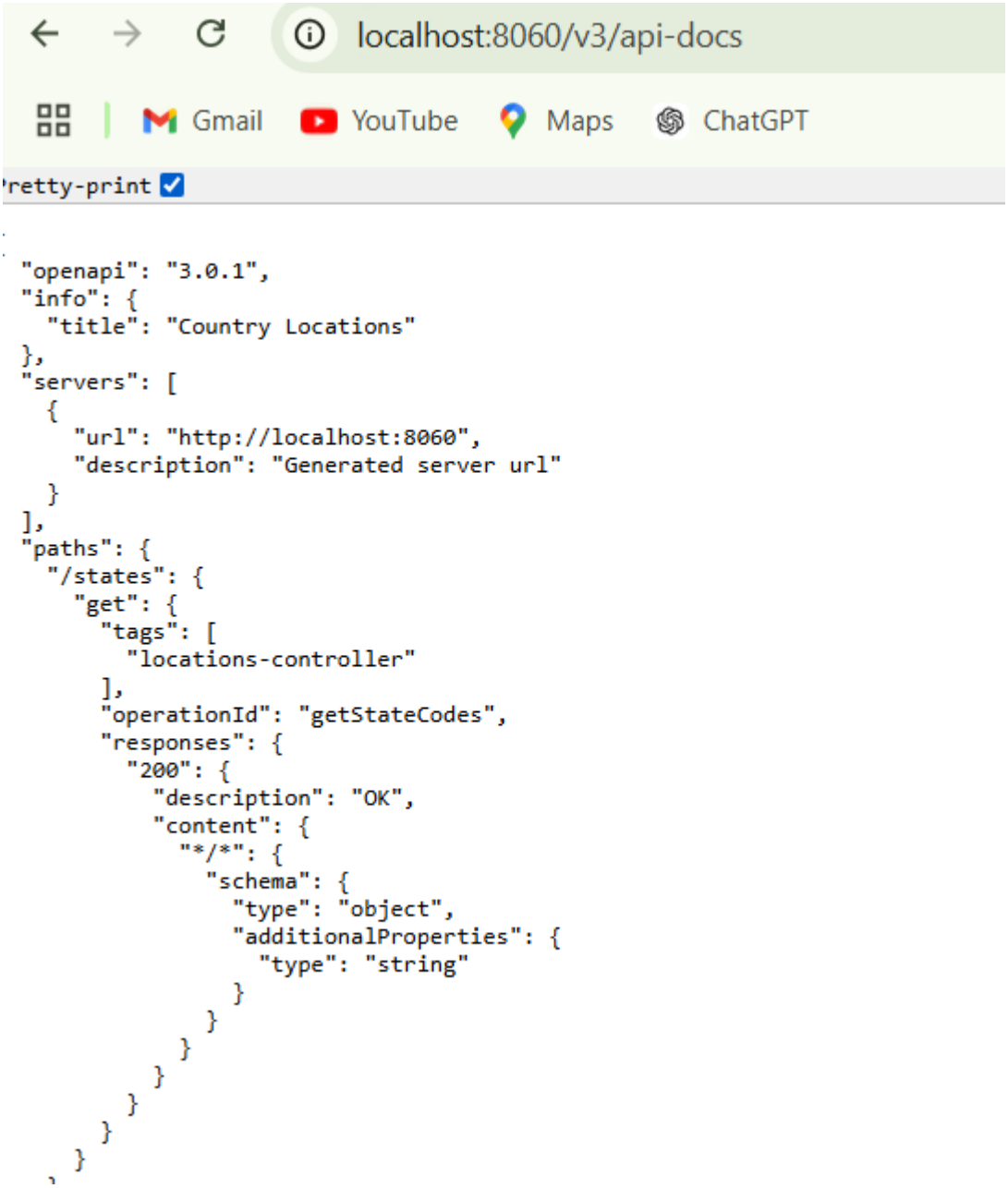
```java

    @Operation(summary = "Retrieve State Codes", description = "Fetches a map of state codes with their corresponding state names.
    @ApiResponse(responseCode = "200", description = "Successfully retrieved the list of state codes")
    @PostMapping("/addCountryCode")
    public ResponseEntity<String> addCountryCode(@RequestParam String code, @RequestParam String name) {
        if (countryCodes.containsKey(code)) {
            return ResponseEntity.badRequest().body("Country code already exists.");
        }
        countryCodes.put(code, name);
        return ResponseEntity.ok("Country added successfully.");
    }

    @PutMapping("/updateCountryCode")
    public ResponseEntity<String> updateCountryCode(@RequestParam String code, @RequestParam String name) {
        if (!countryCodes.containsKey(code)) {
            return ResponseEntity.badRequest().body("Country code does not exist.");
        }
        countryCodes.put(code, name);
        return ResponseEntity.ok("Country updated successfully.");
    }

    @DeleteMapping("/removeCountryCode")
    public ResponseEntity<String> removeCountryCode(@RequestParam String code) {
        if (!countryCodes.containsKey(code)) {
            return ResponseEntity.badRequest().body("Country code does not exist.");
        }
        countryCodes.remove(code);
        return ResponseEntity.ok("Country removed successfully.");
    }

    @GetMapping("/getStateCodes")
    public ResponseEntity<Map<String, String>> getStateCodes() {
        return ResponseEntity.ok(stateCodes);
    }


    @PostMapping("/addStateCode")
    public ResponseEntity<String> addStateCode(@RequestParam String code, @RequestParam String name) {
        if (stateCodes.containsKey(code)) {
            return ResponseEntity.badRequest().body("State code already exists.");
        }
        stateCodes.put(code, name);
        return ResponseEntity.ok("State added successfully.");
    }

    @PutMapping("/updateStateCode")
    public ResponseEntity<String> updateStateCode(@RequestParam String code, @RequestParam String name) {
        if (!stateCodes.containsKey(code)) {
            return ResponseEntity.badRequest().body("State code does not exist.");
        }
        stateCodes.put(code, name);
        return ResponseEntity.ok("State updated successfully.");
    }

    @DeleteMapping("/removeStateCode")
    public ResponseEntity<String> removeStateCode(@RequestParam String code) {
        if (!stateCodes.containsKey(code)) {
            return ResponseEntity.badRequest().body("State code does not exist.");
        }
        stateCodes.remove(code);
        return ResponseEntity.ok("State removed successfully.");
    }


```

## Step 4: Run the Application

1. Start the Spring Boot application.
2. Open a browser and go to: **http://localhost:8080/swagger-ui/index.html**



3. **Or [http://localhost:8060/v3/api-docs](http://localhost:8060/v3/api-docs)**





```
"openapi": "3.0.1",
"info": {
  "title": "Country Locations"
},
"servers": [
  {
    "url": "http://localhost:8060",
    "description": "Generated server url"
  }
],
"paths": {
  "/states": {
    "get": {
      "tags": [
        "locations-controller"
      ],
      "operationId": "getStateCodes",
      "responses": {
        "200": {
          "description": "OK",
          "content": {
            "*/*": {
              "schema": {
                "type": "object",
                "additionalProperties": {
                  "type": "string"
                }
              }
            }
          }
        }
      }
    }
  }
```


You will see the Swagger UI automatically generated by Springdoc, with endpoints from your REST controller.

## Step 5: Customize API Documentation

### 1. Group APIs with @Tag

Use @Tag at the class level to describe and group APIs.

```
@Tag(name = "Country API", description = "**APIs for managing country codes**")
```

**2. Describe Operations with @Operation**

Use @Operation at the method level for summaries and descriptions.

```
@Operation(summary = "Get all country codes", description = "Returns a list of country cod
```

**3. Add Response Descriptions with @ApiResponse**

Add expected responses for better documentation.

```
@ApiResponses(value = {
    @ApiResponse(responseCode = "200", description = "Success"),
    @ApiResponse(responseCode = "404", description = "Country not found")
})
```

```
@DeleteMapping("/countries/{code}")
@Operation(summary = "Delete a country code")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200", description = "Country deleted successfully"),
    @ApiResponse(responseCode = "404", description = "Country code not found")
})
public ResponseEntity<String> deleteCountry(@PathVariable String code) {
    // Method logic
}
```

**4. Define Request Parameters with @Parameter**

Describe query and path parameters.

```
@Parameter(description = "The country code to add")
@RequestParam String code
```

## Step 7: Additional Features

**1. Add API Info in** application.properties

Configure general metadata for your API:

```
springdoc.api-docs.title=Country Management API
springdoc.api-docs.version=1.0.0
springdoc.api-docs.description=APIs for managing and retrieving country codes.
```

**2. Customizing Swagger Path**

If you want to access Swagger on a custom path:

```
springdoc.swagger-ui.path=/swagger
```